DS 4400: Machine Learning and Data Mining 1

Spring 2022
Project Report

Title: Clothing Image Classification

TA: Nate Hoffman

Team: Rebecca Dashevsky

Code: https://colab.research.google.com/drive/1kv17GZU2Ovlh2v-IlS-UOl5wbYqpd5oU?usp=sharing

Video: https://www.canva.com/design/DAE_enGkLGU/cRvNSj0QG-7-1ECrJiPffg/view?utm_content=DAE_enGkLGU&utm_campaign=designshare&utm_medium=link&utm_source=recording_view

Problem Description:

I decided to explore an image classification task using Fashion MNIST data. With this data we can predict the type of clothing. I decided to choose this as a project idea since I am interested in the fashion industry, so I wanted to do something that was applicable to this industry. The goal of this project is to successfully classify the types of clothing we have in our dataset. Another version of this problem could be brand classification- but this data is harder to obtain than the issue I chose to tackle (would need to get product images from a set of brands and label them by hand). As I switched my project idea after the first proposal- I was not able to obtain said data. With what we have, this could be useful for a reselling entity like 'Grailed', a website that lets anyone buy and sell their clothing. For example- if hundreds of people upload products in a day, a tool like this could be used to sift through the uploaded products and classify them for what category they should go in on the website. (https://www.grailed.com/).

As the fashion industry and data continue to cross paths, more tasks like such will be useful to companies and consumers.

References:

1. https://scikitlearn.org/stable/auto_examples/model_selection/plot_randomized_search.html
2. https://towardsdatascience.com/multilayer-perceptron-for-image-classification-5c1f25738935

3. https://michael-fuchs-python.netlify.app/2021/02/03/nn-multi-layer-perceptron-classifier-mlpclassifier/#mlpclassifier
4. https://scikit-learn.org/stable/modules/neural_networks_supervised.html
5. https://machinelearningmastery.com/multinomial-logistic-regression-with-python/
6. https://www.tensorflow.org/tutorials/images/cnn
7. https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83
8. https://github.com/zalandoresearch/fashion-mnist
9. http://mmlab.ie.cuhk.edu.hk/projects/DeepFashion.html
10. https://towardsdatascience.com/understanding-and-using-k-nearest-neighbours-aka-knn-for-classification-of-digits-a55e00cc746f
11. https://www.analyticsvidhya.com/blog/2021/06/mnist-dataset-prediction-using-keras/
12. https://faiss.ai/
13. https://medium.com/criteo-engineering/introducing-autofaiss-an-automatic-k-nearest-neighbor-indexing-library-at-scale-c90842005a11
14. https://medium.com/@plog397/auc-roc-curve-scoring-function-for-multi-class-classification-9822871a6659
15. https://towardsdatascience.com/exploratory-data-analysis-ideas-for-image-classification-d3fc6bbfb2d2

Dataset

The data set contains 60k training, and a 10k test set. The data is represented as 28x28 arrays representing the pixels of the image. Since all datums were a value 0-255, these were scaled [0,1] for performance. The data was imported using the Keras API. (https://keras.io/api/datasets/fashion_mnist/)

The categories/labels are: 0: T-shirt, 1: Trouser, 2: Pullover, 3: dress, 4: coat, 5: sandal, 6: shirt, 7:sneaker, 8: bag, 9: ankle boot

Some example images:

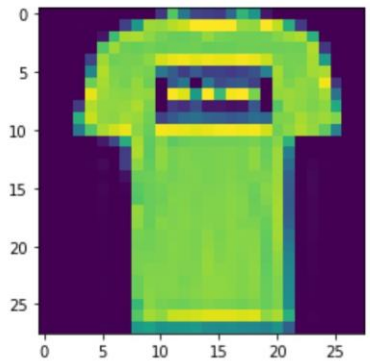


Figure 1:shirt

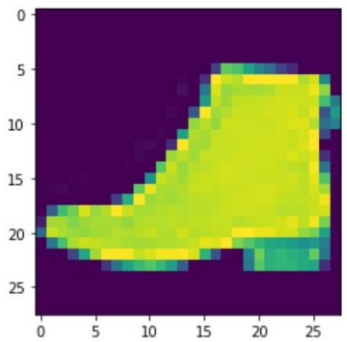


Figure 2: boot

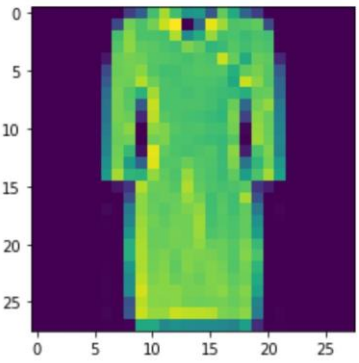
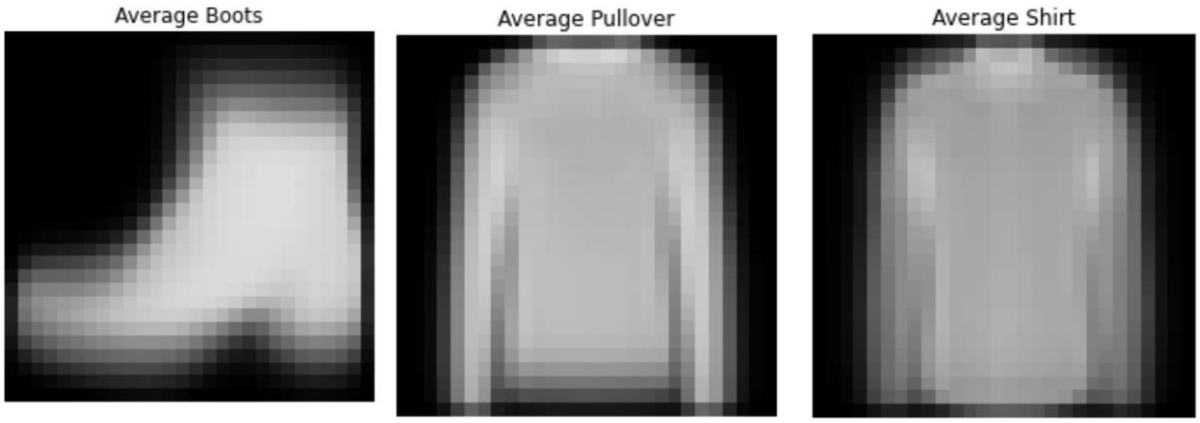


Figure 3:dress

Averages:



Above are images of the 'average' photo of a said clothing item. From here we see that shirt and pullover are almost identical. We can expect our models to have some trouble classifying these, as they look very similar even to a human eye.

Below we can see there is a perfectly even distribution with all the classes. The barplot shown below counts frequencies in the y_train set 60k samples, so the labels are perfectly divided.

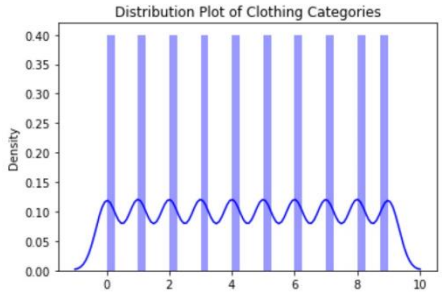


Figure 4: Evenly distributed labels

The data overall did not need that much pre-processing, besides scaling and reshaping as needed for the models, this either meant a flat array of 784, or padding with 0s to get a better CNN result.

Approach and Methodology

As this is a classification problem- we are looking for the best model to correctly classify images of clothing. The features of this model are derived from the pixels of the image. Since this data was already preprocessed into 28 by 28 matrices- our features for some of the models is simply a 784 long array, each xi being a pixel on the image with its intensity.

The data was imported and split 75% for training- 60k records, and 25% for the testing- 10k records. This was done using sklearn train_test_split. After the data was split- we needed to do various things to the training array before it goes into the models, of example flattening the pixels into the size 784 array.

This data was then trained on the models below:

KNN- (Faiss)

For a KNN model it was necessary to reshape the 28 by 28 pixels into a 784 array. This way- each image had 784 x values associated to it. Obviously, it is not good to have such a high number of features, which is why this is just a preliminary analysis. I was curious to see how this data would behave for this model.

The results were not necessarily bad- mostly in the 90s (for each class), but hopefully we can improve the weaker classes like shirt, coat, and pullover. It makes sense to a human eye why these get confused, even more so why we need a deep learning model to make even finer grained distinctions between something like a shirt/coat/pullover.

After poor luck with tuning normal KNN, I came across the Faiss package. Faiss is another version of KNN that works much faster than the original. Using this package, I was able to choose k-neighbors to be 4, and look at some metrics for that.

Faiss is Facebook AI Similarity search, which is not the same as knn in terms of distance, but it does look at the overall similarity between the vectors, which is much more suitable for sparse data we had.

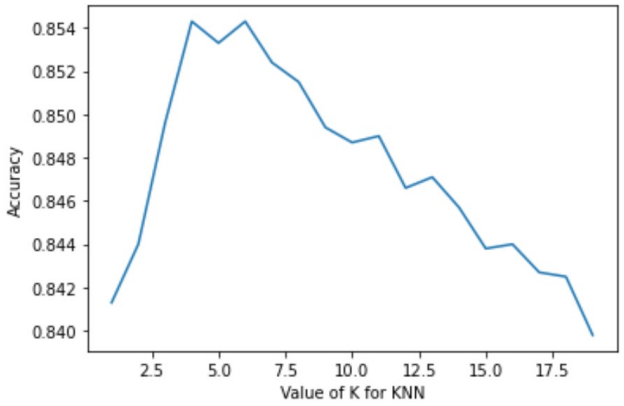


Figure 5: choosing a value of K for Faiss Implementation

MultiNomial Logistic Regression:

As another basic model- I wanted to try multinomial logistic regression. For this model- the tuning parameters are the max number of iterations, and there are different options for penalty with sklearns packaged version. This model used the flattened 784 array as input. It was found that lbfgs was the best solver for this case- as other solvers were not able to run in a timely manner. It was found that convergence was very difficult to obtain- even up to 3000 iterations convergence was still not found- but this did have an accuracy of around .83. Due to this result- we can assume that this data is not linearly separable.

SGD Classifier:

A softmax classifier was used as well for the class predictions. This model uses gradient descent to find the optimal theta to use for predictions. The data used for the model was the same as the input for KNN- which is a flattened 784 long array containing information about each pixel.

Parameter tuning: After failed attempts of gridSearch search due to the volume of the data, and also an attempt of RandomizedSearch, I chose to go about picking the hyper parameters in a more isolated fashion. Gridsearch took far too long to run- but randomsearch did not give a better result than the isolated fashion.

```
RandomizedSearchCV took 8109.36 seconds for 15 candidates parameter settings.
Model with rank: 1
Mean validation score: 0.822 (std: 0.003)
Parameters: {'alpha': 0.053360561210244496, 'average': False, 'l1_ratio': 0.014919115419067541}

Model with rank: 2
Mean validation score: 0.817 (std: 0.004)
Parameters: {'alpha': 0.03058457662435437, 'average': False, 'l1_ratio': 0.07517144367277018}

Model with rank: 3
Mean validation score: 0.814 (std: 0.005)
Parameters: {'alpha': 0.011043319746202826, 'average': False, 'l1_ratio': 0.23535042580826215}
```

Figure 6: randomSearch for SGD

Isolated Search: running I was getting errors that 50 and 100 up to 1000 were not enough, as the model never found convergence for smaller values. So I decided to only look at higher values 1000- and did another test for the alpha with the max iterations set to 1000.

Now for 1000 iterations- it seems an alpha of .01 is the best learning rate.

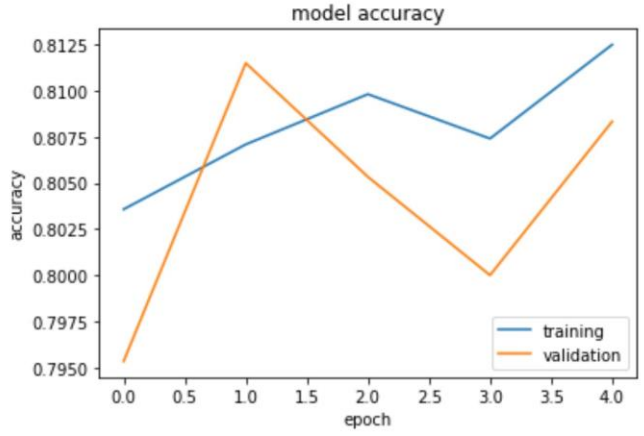
```
for learning rate: 0.01
Accuracy:  0.8236
for learning rate: 0.1
Accuracy:  0.8183
for learning rate: 1
Accuracy:  0.779
for learning rate: 0.5
Accuracy:  0.794
```

the final sgd classifer used 1000 iterations, and a learning rate of .01. (not pictured- a lower learning rate of .001 was also tired but had an accuracy close to 60%)

Multi Layer Perceptron:

For the MLP, the data needed was a scaled 784 long array.

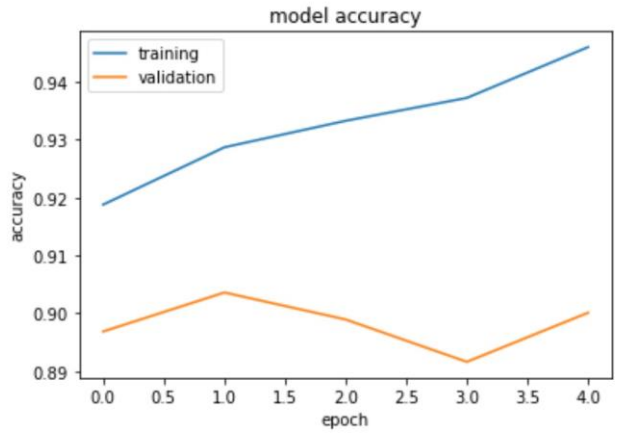
For the first try, I just started off by running a simple 2 layer model, and slightly tweaking some of the inputs to the layers using Keras.




```
Test loss: 0.592
Test accuracy: 0.797
```

Figure 7: the first run

Parameter tuning included adjusting the number of units- I wanted to stray away from adding layers as in my research I found that adding more layers to when working with MNIST usually had adverse effects. Another change that was done was trying different optimizers, the adam optimizer seemed to work much better than the original sgd. Changing the activation did not have much effect on the model- it was kept at relu.




```
Test loss: 0.432
Test accuracy: 0.882
```

Figure 8: 512 units, relu activation, 2 layers

I also tried adding more layers- and seeing the effects. For the most part the accuracy was around .86-.88.

Layer (type)	Output Shape	Param #
dense_52 (Dense)	(None, 2048)	1607680
dense_53 (Dense)	(None, 1024)	2098176
dense_54 (Dense)	(None, 512)	524800
dense_55 (Dense)	(None, 128)	65664
dense_56 (Dense)	(None, 10)	1290

```
Total params: 4,297,610
Trainable params: 4,297,610
Non-trainable params: 0
```

Figure 9: this model had an accuracy of .889 on the test data

The above were done with keras-

This model was a little more challenging to tune, as I was just going about a sort of 'brute force' method of seeing which slight changes might increase our accuracy. There are much more complex ways of doing so but due to computational time and power these were not implemented.

CNN: various architectures

Since this is a convolutional network, this was the only model that did not require a flattened array as input. First was tested just a simple CNN, this architecture was a basic architecture I found through research for a CNN model. The last epoch's performance outdid all of the previous models- and this was just an initial trial.

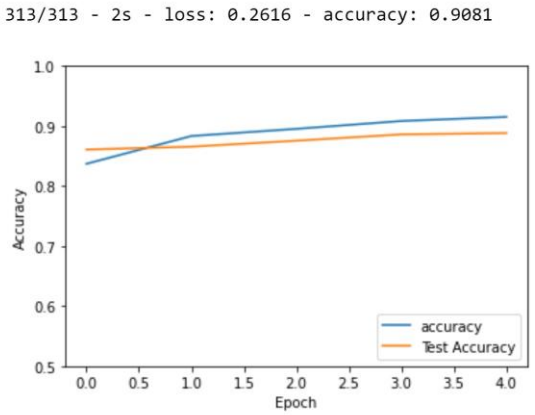


Figure 10: first try with a simple CNN model

I knew it would be possible to get a much better result, so I also wanted to implement one of the popular architectures we went over in class – LeNet. Using this meant I had to pad the train and test sets with 0s in order for the shape to be 32 by 32 instead of 28, which was not done for the first implementation. After the original implementation of LeNet- I got almost the same accuracy as above.

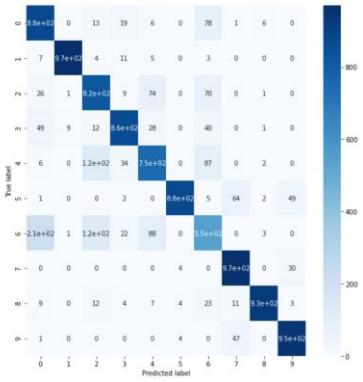
Discussion and Result Interpretation

With the above analysis, we can see that CNN is the best model to use to predict what clothing item an image might be. All the more basic models, not really suitable for image prediction, had accuracies in the low .80s, up to .83. The more complex models did have a significant jump in their accuracy, with the FFNN at high 80s and finally the CNN getting into the 90s range.

KNN: (FNN)

Using the FNN version of FNN, we obtained fairly decent result. This optimized version of KNN runs incredibly faster than KNN, so it was possible to tune and use. From the results we can see that still this model is not suitable for the task- as we have some classes with an F1 under .6.

```
Training Scores:
        Accuracy: 0.91095
        Error: 0.08904999999999996
        F1 Score: 0.91095
        AUC Score: 0.9505277777777777


Testing Scores:
        Accuracy: 0.8543
        Error: 0.14570000000000005
        F1 Score: 0.8543
        AUC Score: 0.9190555555555557
```



	precision	recall	f1-score	support
0	0.74	0.88	0.80	1000
1	0.99	0.97	0.98	1000
2	0.74	0.82	0.78	1000
3	0.90	0.86	0.88	1000
4	0.78	0.75	0.77	1000
5	0.99	0.88	0.93	1000
6	0.64	0.55	0.59	1000
7	0.89	0.97	0.92	1000
8	0.98	0.93	0.95	1000
9	0.92	0.95	0.93	1000
accuracy			0.85	10000
macro avg	0.86	0.85	0.85	10000
weighted avg	0.86	0.85	0.85	10000

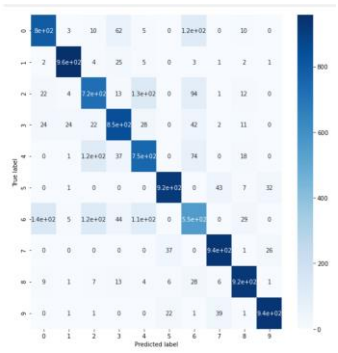
Figure 11: Eval Metrics FNN (CR and Classification report on testing data only)

MultiNomial Logistic Regression:

This was the best of the 'simple' models. This is because the tuning was not too extensive- as there was not much to tune besides just the number of iterations. Even though I was not able to reach a convergence, the accuracy was still comparable to that of SGDClassifier- which was much more complicated to put together. Although one drawback being the model took over 10 minutes just to fit to the data but predicting was very fast. Below we can see that there are still some classes getting confused- like 6 (shirt), 4(coat), and 2(pullover) for the most part.

Classification report on the test data-

	precision	recall	f1-score	support
0	0.80	0.80	0.80	1000
1	0.96	0.96	0.96	1000
2	0.72	0.72	0.72	1000
3	0.81	0.85	0.83	1000
4	0.73	0.75	0.74	1000
5	0.93	0.92	0.93	1000
6	0.61	0.55	0.58	1000
7	0.91	0.94	0.92	1000
8	0.91	0.93	0.92	1000
9	0.94	0.94	0.94	1000
accuracy			0.83	10000
macro avg	0.83	0.83	0.83	10000
weighted avg	0.83	0.83	0.83	10000



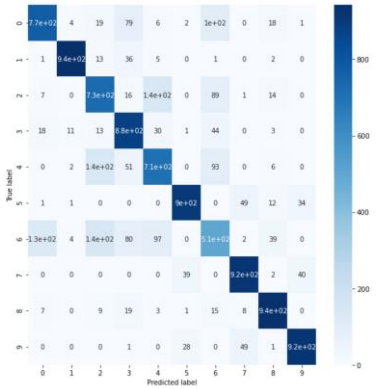
```
Training Scores:
        Accuracy: 0.8873333333333333
        Error: 0.11266666666666669
        F1 Score: 0.8873333333333333
        AUC Score: 0.9374074074074074


Testing Scores:
        Accuracy: 0.8343
        Error: 0.16569999999999996
        F1 Score: 0.8343
        AUC Score: 0.9079444444444444
```

Figure 12: MLR classification report *Figure 13: confusion matrix for mLR* *testing and training evaluation metrics*

SGDClassifer:

The tuning of the SGDClassifier was the most difficult of all the classifiers, mostly due to run time- and still not producing that good of results. As I attempted both a RandomSearch and just isolated parameters- both gave me different results but still an accuracy around .83 for both methods of tuning. Compared to multinomial logistic regression- this one is worse, as it performs around the same- but there are many more parameters that could be tuned. In this model again- the same troublesome classes had issues- shirt, pullover, and coat.

	precision	recall	f1-score	support
0	0.83	0.77	0.80	1000
1	0.98	0.94	0.96	1000
2	0.68	0.73	0.71	1000
3	0.76	0.88	0.81	1000
4	0.71	0.71	0.71	1000
5	0.93	0.90	0.91	1000
6	0.60	0.51	0.55	1000
7	0.89	0.92	0.91	1000
8	0.91	0.94	0.92	1000
9	0.92	0.92	0.92	1000
accuracy			0.82	10000
macro avg	0.82	0.82	0.82	10000
weighted avg	0.82	0.82	0.82	10000



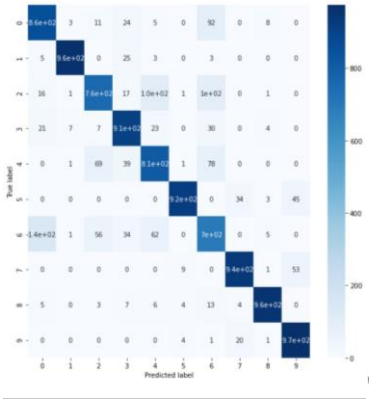
Training Scores:
Accuracy: 0.8352333333333334
Error: 0.16476666666666662
F1 Score: 0.8352333333333334
AUC Score: 0.9084629629629628

Testing Scores:
Accuracy: 0.8214
Error: 0.17859999999999998
F1 Score: 0.8214
AUC Score: 0.9007777777777777

Figure 14: classification report *confusion matrix on test data* *training and testing evaluation metrics*

MLP:

The multi-layer perceptron was the 3rd model to be tuned and tested. This consistently ran significantly faster than all the previous models. With minimal tuning, I was able to achieve accuracies up to .9 at the highest. Unfortunately, this was not much better than the pervious- much slower models. Interestingly, the MLP had the best performance on the infamous 'shirt'.



	precision	recall	f1-score	support
0	0.82	0.86	0.84	1000
1	0.99	0.96	0.98	1000
2	0.84	0.76	0.80	1000
3	0.86	0.91	0.88	1000
4	0.80	0.81	0.81	1000
5	0.98	0.92	0.95	1000
6	0.69	0.70	0.69	1000
7	0.94	0.94	0.94	1000
8	0.98	0.96	0.97	1000
9	0.91	0.97	0.94	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

Training Scores:
Accuracy: 0.91255
Error: 0.08745000000000003
F1 Score: 0.91255
AUC Score: 0.9514166666666666

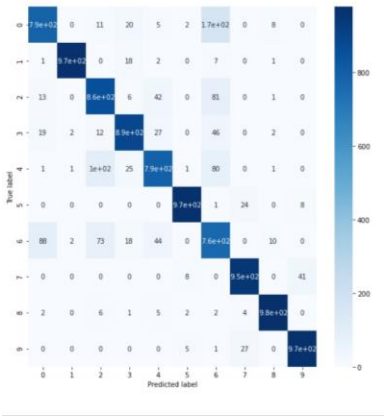
Testing Scores:
Accuracy: 0.8772
Error: 0.12280000000000002
F1 Score: 0.8771999999999999
AUC Score: 0.9317777777777776

Figure 15: Confusion matrix *classification report on Test data* *Testing and training evaluation metric*

CNN:

With CNN the commonly misclassified items are still misclassified- but a little less than the prior models. 6 (shirt) still has the worst performance, but slightly better than other models. It is clear why CNN models are best for image classification.

	precision	recall	f1-score	support
0	0.86	0.79	0.82	1000
1	0.99	0.97	0.98	1000
2	0.81	0.86	0.83	1000
3	0.91	0.89	0.90	1000
4	0.86	0.79	0.83	1000
5	0.98	0.97	0.97	1000
6	0.66	0.77	0.71	1000
7	0.95	0.95	0.95	1000
8	0.98	0.98	0.98	1000
9	0.95	0.97	0.96	1000
accuracy			0.89	10000
macro avg	0.90	0.89	0.89	10000
weighted avg	0.90	0.89	0.89	10000



Training Scores:
Accuracy: 0.91255
Error: 0.08745000000000003
F1 Score: 0.91255
AUC Score: 0.9514166666666666

Testing Scores:
Accuracy: 0.8772
Error: 0.12280000000000002
F1 Score: 0.8771999999999999
AUC Score: 0.9317777777777776

Overall:

CNN is the best choice for image classification. We can see that the commonly confused articles are different types of tops (t-shirts vs shirts vs pullovers). From this information if the project was continued, we might want to consider a more refined methodology on the confused clothing items. A common issue throughout the course of this evaluation was the sheer amount of time it takes to run the non-Neural Network models on image data- as the amount of features is quite large, a reminder that CNN is the best for images for many reasons.

Conclusion:

This project could have many potential applications in the fashion industry- which has made this an interesting choice of problems. There could be many adaptations to this- and using different datasets of clothing from different brands. If there was more time for this project- I would have loved to have obtained my own clothing data- and done a bit of a different prediction, such as brand. This type of prediction could be useful in the same ways as clothing item prediction- and going even deeper it could become a problem of bootleg product detection.

As for the model performance- all of the models struggled with the same 3 classes, shirt, pullover and coat. To a human- it makes sense how these could get confused as they do look visually similar. None of the models successfully could really tell the difference between these, and in a further search we may want to focus on images of these and have more samples of these and less of more obviously different pieces (like boots and bags).

The biggest takeaway for this project was the amount of runtime it took to get models properly working. This was a valuable learning experience and required a lot of trial and error. In the future- I know things like not to use a Gridsearch on massive data. Along with the importance of scaling- something I failed to do for the first iterations of testing and suffered horrible run times. It was interesting to compare the runtime of the deep learning algorithms to the simpler ones. CNN and MLP training were almost instant, compared to the immense amount of time it took to train and tune the SGD classifier, which did not even produce that good of a result. It is

very clear to me now, from experience, what a major impact CNN has had on machine learning, as this was the best model to train and fit for many reasons. It was quick to train, and accurate, the other simpler models could have been omitted, but it was a good learning experience of what not to do with image data.